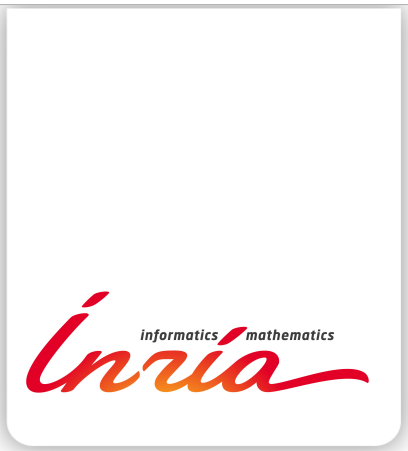
Budget-aware scheduling algorithms for scientific workflows on IaaS cloud platforms

Yves Caniou, Eddy Caron, Aurélie Kong Win Chang, Yves Robert

RESEARCH REPORT
N° 9088
August 2017
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9088--FR+ENG



Budget-aware scheduling algorithms for scientific workflows on IaaS cloud platforms

Yves Caniou*, Eddy Caron*, Aurélie Kong Win Chang*, Yves Robert*†

Project-Team ROMA

Research Report n° 9088 — August 2017 — 27 pages

Abstract: This report introduces several budget-aware algorithms to deploy scientific workflows on IaaS Cloud platforms, where users can request Virtual Machines (VMs) of different types, each with specific cost and speed parameters. We use a realistic application/platform model with stochastic task weights, and VMs communicating through a datacenter. We extend two well-known algorithms, HEFT and Min-Min, and make scheduling decisions based upon machine availability *and* available budget. During the mapping process, the budget-aware algorithms make conservative assumptions to avoid exceeding the initial budget; we further improve our results with refined versions that aim at re-scheduling some tasks onto faster VMs, thereby spending any budget fraction left-over by the first allocation. These refined variants are much more time-consuming than the former algorithms, so there is a trade-off to find in terms of scalability. We report an extensive set of simulations with workflows from the Pegasus benchmark suite. Budget-aware algorithms generally succeed in achieving efficient makespans while enforcing the given budget, and despite the uncertainty in task weights.

Key-words: workflow, checkpoint, fail-stop error, resilience.

* LIP, École Normale Supérieure de Lyon, CNRS & Inria, France
† Univ. Tenn. Knoxville, USA

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Algorithmes d'ordonnancement avec contrainte de budget pour l'exécution de workflows scientifiques sur plates-formes de type *IaaS cloud*

Résumé : Ce rapport présente plusieurs algorithmes prenant en compte le budget pour déployer des workflows scientifiques sur des plateformes de Cloud de type IaaS, sur lesquelles les utilisateurs peuvent utiliser des machines virtuelles (ou *Virtual Machines*, VMs) de différents types, ces dernières étant caractérisées par un coût et une vitesse qui leur sont propres. Nous utilisons un modèle de plateforme et de workflow réalistes avec des tâches de taille stochastique et des VMs communiquant par le biais d'un datacenter. Nous étendons deux algorithmes connus, HEFT et Min-Min, et effectuons l'ordonnancement en nous basant à la fois sur la disponibilité des machines *et* le budget disponible. Pendant le processus d'attribution des VMs aux tâches, les algorithmes prenant le budget en compte se basent sur des hypothèses conservatives afin d'éviter de dépasser le budget initial ; nous améliorons nos résultats en proposant des algorithmes raffinant ces solutions en tentant de ré-assigner certaines taches à des VMs plus rapides, en utilisant pour ce faire la part de budget restant suite à l'ordonnancement initial. Ces versions raffinées demandent plus de temps que les algorithmes proposés plus tôt, il y a donc un compromis à faire en termes de scalabilité. Nous présentons un vaste ensemble de simulations effectuées sur des workflows obtenus à l'aide d'un logiciel de benchmark de Pegasus. Les algorithmes prenant en compte le budget réussissent en général à obtenir des makespans efficaces tout en respectant le budget accordé, et ce malgré l'incertitude concernant la taille des tâches.

Mots-clés : workflow, ordonnancement, HEFT, Min-Min, contrainte de budget, IaaS cloud.

1 Introduction

IaaS (Infrastructure as a Service) Cloud platforms provide a convenient service to many users. Many vendors provide commercial offers with various characteristics and price policies. In particular, a large choice of VM (Virtual Machine) types is usually provided, that ranges from slow-but-cheap to powerful-to-expensive devices. When deploying a scientific workflow on an IaaS cloud, the user is faced with a difficult decision: which VM type to select for which task? How many VMs to rent? These decisions clearly depend upon the budget allocated to execute the workflow, and are best taken when some knowledge on the task profiles in the workflow is available. The standard practice is to run a classical scheduling algorithm, whether HEFT [17] or MIN-MIN [3, 9], with a VM type selected arbitrarily, and to hope for the best, *i.e.,* that the budget will not be exceeded at the end. To remedy such an inefficient approach, this paper introduces several budget-aware algorithms to deploy scientific workflows on IaaS clouds. The main idea is to revisit well-known algorithms such as HEFT and MIN-MIN and to make a decision for each task to be scheduled based upon both machine availability *and* remaining budget.

While several cost-aware algorithms have been introduced in the literature (see Section 2 for an overview), this paper makes new contributions along the following lines:

- A realistic application model, with stochastic task weights;
- A detailed yet tractable platform model, with a datacenter and multiple VM categories;
- Budget-aware algorithms that extend HEFT and MIN-MIN, two widely-used list-scheduling algorithms for heterogeneous platforms;
- Refined (but more costly) variants that squeeze the most of any leftover budget to further decrease total execution time. The refined versions aim at exploiting the opportunity to re-schedule some tasks onto faster VMs, thereby spending any budget fraction leftover by the first allocation. These refined variants are much more time-consuming than the former algorithms, so there is a trade-off to find in terms of scalability.

The rest of the paper is organized as follows. We introduce the performance model in Section 3. We describe budget-aware scheduling algorithms in Section 4: Section 4.1 presents the extensions to HEFT and MIN-MIN, while Section 4.2 provides the refined versions. Section 5 is devoted to assessing their performance through extensive simulations. Finally, we provide concluding remarks and directions for future work in Section 6.

2 Related work

Many scientific applications from various disciplines are structured as workflows [2]. Informally, a workflow can be seen as the composition of a set of basic operations that have to be performed on a given input data set to produce the expected scientific result. The development of complex middleware with workflow engines [6, 7, 4] has automated workflow management. IaaS Clouds raised a lot of interest recently, thanks to an elastic resource allocation and pay-as-you-go billing model. In a Cloud environment, there exist many solutions for scheduling workflows [11, 16], some of which include data management strategies [18]. Also, [13] introduced two auto-scaling mechanisms to solve the problem of the resource allocation in a cost-efficient way for unpredicted workflow jobs. [19] introduced a workflow scheduling in Clouds solutions with security and cost considerations. [1] provides guidelines and analysis to understand cost optimization in scientific workflow scheduling by surveying existing

approaches in Cloud computing.

To the best of our knowledge, the closest paper to this work is [12], which proposes workflow scheduling algorithms under both budget and deadline constraints. Their platform model is similar to ours, although we allow for computation/transfer overlap and account for a startup delay t_{boot} to boot a VM. However, their application framework and objective are different: they consider workflow ensembles, *i.e.,* sets of workflows with priorities, that are submitted for execution simultaneously, and they aim at maximizing the number, or the cumulated priority value, of the workflows that complete successfully under the constraints. Still, we share the approach of partitioning the initial budget into chunks to be allotted to individual candidates (workflows in [12], tasks in this paper).

3 Model

This section details the application and platform model used to assess the performance of the scheduling algorithms. Table 1 summarizes the main notations used in this paper.

Workflows	
n	number of tasks in the workflow
T_i	The i^{th} task of the workflow
$\overline{w_i}$, σ_i	weight of T_i: mean, standard deviation
$size(d_{T_i,T_j})$	size of input file from T_i to T_j
Platform	
k	number of VM categories
$s_1 \leq s_2 \cdots \leq s_k$	VM speeds
$\overline{s}$	average speed
$c_{h,k}$, $c_{ini,k}$	unit cost and initial cost for category k
c_{tsf}	unit cost of I/O operations
$c_{h,DC}$	unit cost of datacenter usage
bw	bandwidth between VMs and datacenter

Table 1: Summary of main notations.

3.1 Workflows

The model of workflows presented here is directly inspired by [10, 12]. A task workflow is represented with a DAG (Directed Acyclic Graph) $G = (V, E)$, where V is the set of tasks to schedule, and E is the set of dependencies between tasks. In this model, a dependency corresponds to a data transfer between two tasks. Tasks are not preemptive and must be executed on a single processor[1]. Most workflow scheduling algorithms use as starting assumption that the exact number of instructions constituting a task is known in advance, so that its execution time is given accurately. However, this hypothesis is not always realistic. The number of instructions for a given task may strongly depend on the current input data, such as in image processing kernels. In our model, we only know an estimation of the number of instructions for each task. For lack of knowledge about the origin of time variations, we

[1]This assumption is only for the sake of the presentation; it is easy to extend the approach to parallel tasks.

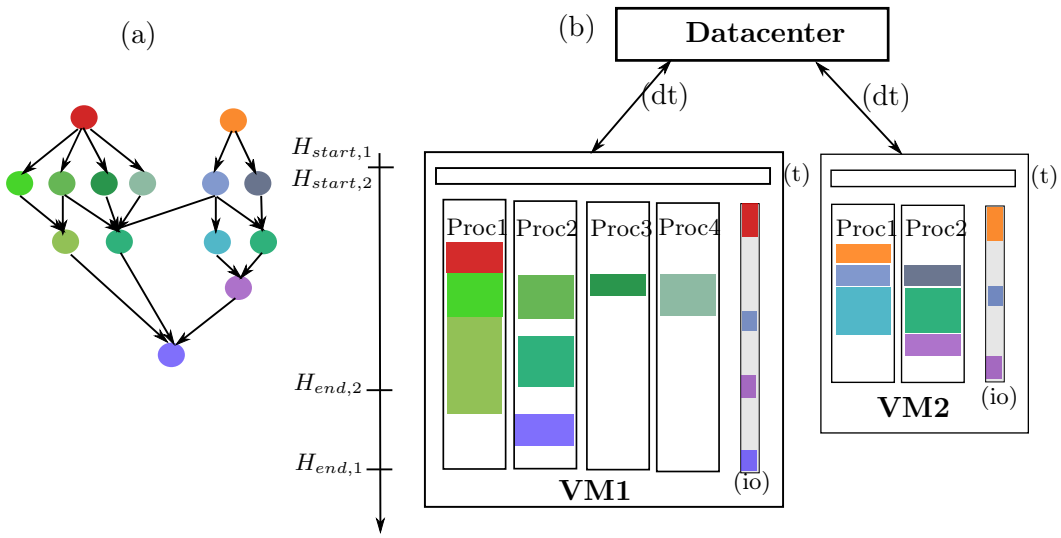


Figure 1: Workflow (a) and platform (b): VMs and their processors, initialization time (t) of duration d_{ini}, interactions with the common datacenter (dt); and example of a schedule with transmission times (io).

assume that all the parameters which determine the number of instructions forming a task are independent. This resulting number is the task *weight* and follows a Gaussian law with mean $\overline{w_i}$ and standard deviation σ_i which can be estimated (for example by sampling).

To each dependency $(T_i, T_j) \in E$ is associated an amount of data of size $size(d_{T_i,T_j})$. We say that a task T is ready if either it does not have any predecessor in the dependency graph, or if all its predecessors have been executed and all the output files generated.

3.2 Platform

Our model of Cloud platform (see Figure 1) mainly consists of a datacenter and processing units. It is based to a great extent on the offers of three big Cloud providers: Google Cloud[2], Amazon EC2[3] and OVH[4]. Given that Cloud providers propose a fault-tolerance service which ensures a very high availability of resources (in general over 99.97%[5]) as well as sufficient data redundancy, the datacenter and processing units are considered reliable and not subject to faults.

There is only one datacenter, used by all processing units. It is the common crossing point for all the data exchanges between processing units: these units do not interact directly, because of security issues for example. When a task T is to be executed on a VM v, each input file of T generated by one predecessor T' must be accessed from the datacenter, unless that this file has been produced on the same VM v (meaning that T' had been scheduled on v too).

The datacenter is also where the final generated files are stored before being transferred to the user. For simplicity, we consider that the datacenter bandwidth is large enough to feed

[2] `https://cloud.google.com/compute/pricing`
[3] `https://aws.amazon.com/ec2/pricing/on-demand/`
[4] `https://www.ovh.com/fr/public-cloud/instances/tarifs/`
[5] `https://cloudharmony.com/status`

all processing units. Likewise, we consider that the datacenter is able to answer all submitted requests simultaneously, without any supplementary cost.

The processing units are VMs (Virtual Machines). They can be classified in different categories characterized by a set of parameters fixed by the provider. Some providers offer parameters of their own, such as the number of forwarding rules[6]. We only retain parameters common to the three providers Google, Amazon and OVH: A VM of category k has n_k processors, one processor being able to process one task at a time; A VM has also a speed s_k corresponding to the number of instructions it can process per time unit, a cost per time-unit $c_{h,k}$ and an initial cost $c_{ini,k}$; All these VMs take an initial, and uncharged, amount of time t_{boot} to boot before being ready to process tasks. Already integrated in the schedule computing process, this starting time is thus not counted in the cost related to the use of the VM, which is presented in Section 3.4. Without loss of generality, categories are sorted according to hourly costs, so that $c_{h,1} \leq c_{h,2} \cdots \leq c_{h,n_k}$. We expect speeds to follow the same order, but do not make such an assumption.

The platform thus consists of a set of n VMs of k possible categories. Some simplifying assumptions make the model tractable while staying realistic: (i) We assume that the bandwidth is the same for every VM, in both directions, and does not change throughout execution; (ii) A VM is able to store enough data for the tasks assigned to it: in other words, a VM will not have any memory/space overflow problem, so that every increase of the total makespan will be because of the stochastic aspect of the task weights; (iii) Initialization time is the same for every VM; (iv) Data transfers take place independently of computations, hence do not have any impact on processor speeds to execute tasks.

We chose an "on-demand" provisioning system: it is possible to deploy a new VM during the workflow execution if needed. Hence VMs may have different start-up times, and thus won't start at the same time. A VM v is started at time $H_{start,v}$ and does not stop until all the files created by its last computed task have been transferred to the datacenter, at time $H_{end,v}$. VMs are allocated by continuous slots. If one wants discontinuous allocations, one may free the VM, then use a new one later, which at least requires sending all the data generated by the last processed task to the datacenter, and loading all predecessor's output files of the first task scheduled on that new VM before execution.

3.3 Workflow execution

Tasks are mapped to VMs and locally executed in the order given by the scheduling algorithm, such as those described in Section 4. Given a VM v, a task is launched as soon as (i) the VM is idle; (ii) all its predecessor tasks have been executed, and (iii) the output files of those predecessors mapped onto other VMs have been transferred to v via the datacenter.

3.4 Cost

Our cost model is meant to represent generic features out of the existing offers from Cloud providers (Google, Amazon, OVH). The total cost of the whole workflow execution is the sum of the costs due to the use of the VMs and of the cost due to the use of the datacenter $\mathcal{C}_{DC}$. The cost C_v of the use of a VM v of category k_v is calculated as follows:

$$C_v = (H_{end,v} - H_{start,v}) \times c_{h,k_v} + c_{ini,k_v} \tag{1}$$

[6] `https://cloud.google.com/compute/pricing`

There is a start-up cost c_{ini,k_v} in Equation (1), and a term c_{h,k_v} proportional to usage duration $H_{end,v} - H_{start,v}$.

The cost for the datacenter is based on a cost per time-unit $c_{h,DC}$, to which we add a transfer cost. This transfer cost is computed with the amount of data transferred from the external world to the datacenter ($\text{size}(d_{in,DC})$), and from the datacenter to the outside world ($\text{size}(d_{DC,out})$). In other words, $d_{in,DC}$ corresponds to files that are input to entry tasks in the workflow, and $d_{DC,out}$ to files that are output from exit tasks. Letting $H_{start,first}$ be the moment when we book the first VM and $H_{end,last}$ be the moment when the files of the last task processed have entirely been sent to the datacenter, we have:

$$\begin{aligned} C_{DC} = (\text{size}(d_{in,DC}) + \text{size}(d_{DC,out})) \times c_{tsf} \\ +(H_{end,last} - H_{start,first}) \times c_{h,DC} \end{aligned} \tag{2}$$

Altogether, the total cost is

$$C_{wf} = \sum_{v \in R_{VM}} C_v + C_{DC}$$

where R_{VM} is the set of booked VMs during the execution.

3.5 Objective function

Given a deadline $\mathcal{D}$ and a budget $\mathcal{B}$, the objective is to fulfil the deadline while respecting the budget:

$$\mathcal{D} \geq H_{end,last} - H_{start,first} \quad \text{and } \mathcal{B} \geq C_{wf} \tag{3}$$

A more complicated objective would be to find the schedule that minimizes the makespan while respecting the budget:

$$\min(H_{end,last} - H_{start,first}) \text{ where } \mathcal{B} \geq C_{wf}$$

4 Scheduling algorithms

This section introduces HEFTBudg and Min-MinBudg, the budget-aware extensions of HEFT [17] and Min-Min [3, 9], two reference scheduling algorithms widely used by the community. Section 4.1 details the main algorithms, which assign a fraction of the remaining budget to the current task to be scheduled, while aiming at minimizing its completion time. Then Section 4.2 provides refined versions of HEFTBudg that squeeze the most of any leftover budget to re-map some tasks to more efficient VMs. This leads to an improvement in the makespan, at the price of a much larger CPU time of the scheduling algorithms. We did not consider the corresponding refinement of Min-MinBudg, because HEFTBudg turned out to be more efficient than Min-MinBudg in our simulations, always achieving a smaller makespan for the same budget.

4.1 HEFTBudg and Min-MinBudg

The budget-aware extensions of HEFT and Min-Min need to account both for the task stochasticity and budget constraint, while aiming at makespan minimization. Coping with task stochasticity is achieved by adding a certain quantity to the average task weight so that

the risk of under-estimating its execution time is reasonably low, while retaining an accurate value for most executions. We went for a somewhat conservative value for the weight of a task T, namely $\overline{w_T} + \sigma_T$.

Algorithm 1 Dividing the budget into tasks.

1: **function** DIVBUDGET($wf, \mathcal{B}_{calc}, \overline{s}, bw$)
2: $W_{max} \leftarrow$ getMaxTotalWork(wf)
3: $d_{max} \leftarrow$ getMaxTotalTransfData (wf)
4: **for each** T **of** wf **do**
5: $budgPTsk[T] \leftarrow \mathcal{B}_T \leftarrow \mathcal{B}_{calc} \times \frac{\frac{\overline{w_T}+\sigma_T}{\overline{s}} + \frac{size(d_{pred,T})}{bw}}{\frac{W_{max}}{\overline{s}} + \frac{d_{max}}{bw}}$
6: **end for**
7: **return** $budgPTsk$
8: **end function**

As for the budget, given a workflow wf, we first reserve a fraction to cover the cost of the datacenter and the initialization of the VMs, and then we divide what remains into the tasks of the workflow. Let $\mathcal{B}_{ini}$ denote the initial budget. To estimate the amount to be reserved:

- For the cost of the datacenter, we need to estimate the duration $H_{end,last} - H_{start,first}$ of the whole execution (see Equation (2)). To this purpose, we consider an execution on a single VM of the first (cheapest) category, compute the total duration $W_{max} = \sum_{T \in wf}(\overline{w_T} + \sigma_T)$ and let

$$H_{end,last} - H_{start,first} = \frac{W_{max}}{s_1} + \frac{\text{size}(d_{in,DC}) + \text{size}(d_{DC,out})}{bw} \tag{4}$$

  Altogether, we pay the cost of input/output files several times: with factor c_{tsf} for the outside world, with factor $c_{h,DC}$ for the usage of the datacenter (Equation (4)), and with factor $c_{h,1}$ to transfer the files to and from the unique VM. However, there will be no communication internal to the workflow, since we use a single VM.
- For the initialization of the VMs, we assume a different VM of the first category per task, hence we budget the amount $nc_{ini,1}$.

Combining these two choices is conservative: on one hand we consider a sequential execution, but account only for input and output files with the external world, eliminating all internal transfers during the execution; on the other hand we reserve as many VMs as tasks, ready to pay the price for parallelism, at the risk of spending time and money due to file transfers during the execution. Altogether, we reserve the corresponding amount of budget and are left with $\mathcal{B}_{calc}$ for the tasks.

This reduced budget $\mathcal{B}_{calc}$ is shared among tasks in a proportional way (see Algorithm 1): we estimate how much time $t_{calc,T}$ is required to execute each task T, transfer times included, and allocate the corresponding part $\mathcal{B}_T$ of the budget in proportion to the whole for execution of the entire workflow $t_{calc,wf}$:

$$\mathcal{B}_T = \frac{t_{calc,T}}{t_{calc,wf}} \times \mathcal{B}_{calc} \tag{5}$$

In Equation (5), we use

$$t_{calc,T} = \frac{\overline{w_T} + \sigma_T}{\overline{s}} + \frac{size(d_{pred,T})}{bw}$$

where

$$size(d_{pred,T}) = \sum_{(T',T)\in E} size(d_{T',T}) \tag{6}$$

is the volume of input files of T from all its predecessors. Similarly, we use

$$t_{calc,wf} = \frac{W_{max}}{\overline{s}} + \frac{d_{max}}{bw}$$

where $d_{max} = \sum_{(T_i,T_j)\in E} size(d_{T_i,T_j})$ is the total volume of files within the workflow. Computing weights ($\overline{w_T} + \sigma_T$ and W_{max}) are divided by the mean speed $\overline{s}$ of VM categories, while file sizes ($size(d_{pred,T})$ and d_{max}) are divided by the bandwidth bw between VMs and the datacenter. Again, it is conservative to assume that all files will be transferred, because some of them will be stored in-place inside VMs, so it is here another source of over-estimation of the cost. On the contrary, using the average speed $\overline{s}$ in the estimation of the computing time may lead to an under-estimation of the cost when cheaper/slower VMs are selected.

This subdivided budget is then used to choose the best host for each ready task (see Algorithm 2): the best host for a task T from the platform $\mathcal{P}$ will be the one providing the best EFT (Earliest Finish Time) for T, among those respecting the amount of budget $\mathcal{B}_T$ allocated to T. The platform $\mathcal{P}$ is defined as the set of host candidates, which consists of already used VMs plus one fresh VM of each category. For each host candidate *host*, either already used (set Used_{VM}) or new candidate (set New_{VM}), we first evaluate the time $t_{Exec,T,host}$ needed to have T executed (*i.e.,* transfer of input data and computations) on *host*:

$$t_{Exec,T,host} = \delta_{new} \times t_{boot} + \frac{\overline{w_T} + \sigma_{task}}{s_{host}} + \frac{size(d_{in,T})}{bw} \tag{7}$$

In Equation (7), we introduce the boolean δ_{new} whose value is 1 if $host \in \text{New}_{\text{VM}}$ to account for its startup delay, and 0 otherwise. Also, some input files may already be present if $host \in \text{Used}_{\text{VM}}$, thus we use $size(d_{in,T})$ instead of $size(d_{pred,T})$ (see Equation (6)), defining $d_{in,T}$ as those input files not already present on *host*.

To compute $EFT_{T,host}$, the Earliest Finish Time of task T on host *host*, we account for its Earliest Begin Time $t_{begin,host}$ and add $t_{Exec,T,host}$. Then $t_{begin,host}$ is simply the maximum of the following quantities: (i) availability of *host*; (ii) end of transfer to the datacenter of any input file to T. The latter includes all files produced by a predecessor of T executed on another host; these files have to be sent to the datacenter before being re-emitted to *host*, since VMs do not communicate directly. There is a cost associated to these transfers, which we add to $t_{Exec,T,host} \times c_{h,host}$ to compute the total cost $c_{T,host}$ incurred to execute T on *host*. We do not write down the equation defining $t_{begin,host}$, as it is quite similar to previous ones. Since we already subtracted from the initial budget everything except the cost of the use of the VMs themselves, getBestHost() can safely use $\mathcal{B}_T$ as the upper bound for the budget reserved for task T.

In fact, the algorithm reclaims any unused fraction of the budget consumed when assigning former tasks: this is the role of the variable pot, which records any leftover budget in previous assignments. Finally, HEFTBUDG (see Algorithm 4) and MIN-MINBUDG (see Algorithm 3) are the counterpart of the original HEFT and MIN-MIN algorithms, extended with the provisioning for the budget. For some tasks, getBestHost() will not return the host with smallest ETF, but instead the host with smallest ETF among those that respect the allotted budget. The complexity of HEFTBUDG and MIN-MINBUDG is $O(n + e)p$, where n is the

number of tasks, e is the number of dependence edges, and p the number of enrolled VMs. This complexity is the same as for the baseline versions, except that p is not fixed a priori. In the worst case, $p = O(\max(n, k))$ because for each task, we try all used VMs, whose count is possibly $O(n)$, and k new ones, one per category.

Algorithm 2 Choosing the best host for each ready task.

```
1: function getBestHost(T, budgPTsk[T], P, pot)
2:     B_T ← budgPTsk[T] + pot
3:     // initialisation: new host of cheapest category:
4:     bestHost ← host_{new,1}
5:     minEFT ← EFT_{T,bestHost}
6:     for each host of (Used_VM ∪ New_VM) do
7:         if ((EFT_{T,host} < minEFT) and (c_{T,host} <= B_T)) then
8:             minEFT ← EFT_{T,host}
9:             bestHost ← host
10:            pot ← B_T − c_{T,host}
11:        end if
12:    end for
13:    return bestHost, pot
14: end function
```

4.2 HEFTBudg+ and HEFTBudg+Inv

This section details two refined versions of HEFTBudg. Because of the many conservative decisions taken during the design of the algorithm, it is very likely that not all the initial budget $\mathcal{B}_{ini}$ will be spent by HEFTBudg. In order to refine the solution returned from HEFTBudg, we re-consider each decision taken and try to improve it.

HEFTBudg (just as HEFT) assigns priorities to the tasks based upon their bottom level [17]. Let ListT be the ordered list of tasks by non-decreasing priority, and let selSched denote the scheduling returned by HEFTBudg. The first variant HEFTBudg+ (see Algorithm 5) processes the tasks in the order of ListT, hence in the same order as HEFT and HEFTBudg, while HEFTBudg+Inv used the reverse order.

For both variants, let T be the task currently considered: we then generate new schedules obtained by assigning T on either an already used VM except the one given by selSched, and on a new one for each category. We compute c_{tot} and $t_{calc,wf}$ for each of them, and keep the one which has the shortest makespan and respects the budget.

As mentioned in Section 4.1, HEFTBudg (just as HEFT) has complexity $O(n + e)p$, where $p = O(\max(n, k)$ in the worst case. Both HEFTBudg+ and HEFTBudg+Inv start with a full iteration of HEFTBudg; then, for each task, they try a new host and generate the resulting schedule. Hence their complexity is $O(n(n + e)p$, where $p = O(\max(n, k)$ in the worst case. This is an order of magnitude more CPU demanding than HEFTBudg, which limits their usage to smaller-size workflows.

Algorithm 3 MIN-MINBUDG

function MIN-MINBUDG($wf, \mathcal{B}_{calc}, \mathcal{P}$)
  $\overline{s} \leftarrow calcMeanSpeed(\mathcal{P})$
  $bw \leftarrow getBw(\mathcal{P})$
  $budgPTsk \leftarrow divBudget(wf, \mathcal{B}_{calc}, \overline{s}, bw)$
  $pot, newPot \leftarrow 0$
  while ! $areEveryTasksSched(wf)$ **do**
    $selectedHost \leftarrow null$
    $selectedTask \leftarrow null$
    $minFT \leftarrow -1$
    $readyTasks \leftarrow getReadyTasks(wf)$
    for each T **of** wf **do**
      $host \leftarrow getBestHost(T, budgPTsk[T], \mathcal{P}, newPot)$
      $finishTime \leftarrow EFT_{T,host}$
      if $((minFT < 0)$ **or** $(finishTime < minFT))$ **then**
        $minFT \leftarrow finishTime$
        $selectedTask \leftarrow T$
        $selectedHost \leftarrow host$
        $pot \leftarrow newPot$
      end if
    end for
    $push(selectedTask, tskOrder)$
    $sched[selectedTask] \leftarrow selectedHost$
    $schedule(selectedTask, selectedHost)$
    $update(\text{Used}_{\text{VM}})$
  end while
  return $tskOrder, sched$
end function

5 Simulations

This section provides all the simulation results. We first describe the experimental setup in Section 5.1. Next in Section 5.2, we assess the performance of the main algorithms HEFTBUDG and MIN-MINBUDG, using the standard HEFT and MIN-MIN heuristics as a baseline for comparison. Then in Section 5.3, we proceed to the refined variants HEFTBUDG+ and HEFTBUDG+INV, and quantify their improvement in terms of makespan, as well as their additional cost in terms of CPU time.

5.1 Experimental methodology

We designed a simulator based on SimDag [15], an extension of the discrete event simulator SimGrid [5], to evaluate all algorithms. The model described in Section 3 is instantiated with 3 VM categories and respective costs inspired from the offers by Amazon Cloud, Google Cloud and OVH (see Table 2).

We used three types of workflows from the Pegasus benchmark suite [14, 8]: LIGO, CYBERSHAKE and MONTAGE. Concerning LIGO, most input files have the same (large)

Algorithm 4 HEFTBUDG

```
1: function HEFTBUDG(wf, B_calc, P)
2:     s̄ ← calcMeanSpeed(P)
3:     bw ← getBw(P)
4:     budgPTsk ← divBudget(wf, B_calc, s̄, bw)
5:     ListT ← getTasksSortedByRanks(wf, s̄, bw, lat)
6:     pot, newPot ← 0
7:     for each T of ListT do
8:         host ← getBestHost(T, budgPTsk[T], P, newPot)
9:         pot ← newPot
10:        push(T, tskOrder)
11:        sched[T] ← host
12:        schedule(T, host)
13:        update(Used_VM)
14:    end for
15:    return ListT, sched
16: end function
```

size, only one of them is oversized compared with the others (by a ratio over 100). LIGO consists of a lot of parallel tasks sharing a link to some agglomerative tasks, one agglomerative task per little set; this scheme repeats twice since there is a second subdivision after the first agglomeration. In CYBERSHAKE, half the tasks have huge input files. The workflow itself consists of a first set of tasks generating data in parallel, data which will be used by a directly connected task (one calculating task per generating task). These parallel activities are all linked to two different agglomerative tasks. On the contrary, MONTAGE has plenty highly inter-connected tasks, making parallelisation less easy. The number of instructions of its different kinds of tasks is balanced, as is the size of the exchanged files. For each workflow type, we used the simulator available on the Pegasus website to generate our benchmark, with five different instances per workflow type, and different numbers of tasks: 30, 60 and 90; this leads to $5 \times 3 = 15$ workflows per type.

Each generated workflow is then re-used to generate workflows having the same DAG structure, but with different values for task weights: to this purpose, we keep the original weight for a task T as the mean $\overline{w_T}$ and use $25, 50, 75$ and 100% of that mean number for the standard deviation σ_T. In the figures, each simulation was repeated 25 times and we plot mean values; vertical bars represent standard deviations. Overall, 16500 experiments have been executed per workflow type and algorithm.

5.2 Perfomance of HEFTBUDG and MIN-MINBUDG

In this section, we report and compare results for HEFTBUDG and MIN-MINBUDG against those for the baseline algorithms HEFT and MIN-MIN. We report results for 30, 60 and 90 tasks. Figures 2, 3 and 4 have three rows, one per workflow type, and reports the results as a function of the initial budget: makespan in first column, total cost in second column, and number of VMs in third column. We record the number of used VMs as an indicator of some choices made by the budget-aware algorithms that trade-off increased usage of former VMs vs enrolment of new ones. In the first column, the green dot labeled *min_cost* represents the

Algorithm 5 HEFTBUDG+

```
1: function HEFTBUDG+(wf, B_ini, P)
2:     B_calc ← getBudgCalc(wf, B_ini, P)
3:     ListT, selSched ← HEFTBUDG(wf, B_calc, P)
4:     c_tot , t_calc,wf ← simulate(wf, P, ListT, selSched)
5:     minTimeCalc ← t_calc,wf
6:     for each T of ListT do
7:         for each host of ((Used_VM \ sched[T]) ∪ New_VM) do
8:             sched ← schedule(T, host)
9:             c_tot , t_calc,wf ← simulate(wf, P, ListT, sched)
10:            if ((t_calc,wf < minTimeCalc) and (c_tot < B) ) then
11:                selectedHost ← host
12:                minTimeCalc ← t_calc,wf
13:            end if
14:        end for
15:        selSched[T] ← selectedHost
16:        update(Used_VM)
17:    end for
18:    return ListT, selSched
19: end function
```

mean of the cheapest solutions for the asked schedule. It has been obtained allocating all the tasks on the same host, the cheapest one.

The budget constraint is respected in most cases (see Figures 2b, 2e, 2h, 3b, 3e, 3h, 4b, 4e, 4h; the black line draws the affine function of the initial budget). Exceptions are some instances of LIGO with a budget near from the minimal needed to schedule all the tasks. The explanation is the following: we assumed that the bandwidth of the datacenter would be sufficient for all simultaneous transfers, but SimGrid monitors the available bandwidth, which can become a bottleneck; LIGO has a lot of parallel tasks running concurrently, that may well send huge files at the same time. In those very few cases, we underestimated the time needed to send these files.

Clearly, HEFT and MIN-MIN give the same solution as HEFTBUDG and MIN-MINBUDG respectively, when they are given an infinite initial budget. We see that HEFT and MIN-MIN obtain similar makespans, but HEFT uses more VMs than MIN-MIN: for instance running CYBERSHAKE uses an average of 79 VMs for HEFT vs. 33 for MIN-MIN, 90 vs. 22 for LIGO, 90 vs. 56 for MONTAGE). The cost is thus smaller for more parallel workflows (LIGO), and larger for the other ones (CYBERSHAKE, MONTAGE).

What is the initial budget needed by the budget-aware algorithm to achieve the minimal makespan returned by the baseline version? We see that HEFTBUDG needs a smaller initial budget than MIN-MINBUDG for MONTAGE (see Figures 2g, 3g and 4g), and a similar one for CYBERSHAKE and LIGO (see Figures 2a, 2d, 3a, 3d and 4a, 4d). We refine this analysis for CYBERSHAKE in Figures 5: the difference in minimal budgets decreases sharply with the number of tasks. Results are similar for LIGO. This is due to the graph structure of these workflows: for CYBERSHAKE, increasing the number of tasks leads to workflows with a majority of parallel tasks; for LIGO, it leads to an increasing number of independent short workflows. In both cases, increasing the number of tasks redners the workflow closer to

VM parameters	
Categories	$k = 3$
Setup delay	$t_{boot} = 10$ min
Setup cost	$c_{ini,\ell} = \$2$ for $1 \leq \ell \leq 3$
Category 1 (Slow)	Speed $s_1 = 5.2297$ Gflops Cost $c_{h,1} = \$0.145$ per hour
Category 2 (Medium)	Speed $s_2 = 8.8925$ Gflops Cost $c_{h,2} = \$0.247$ per hour
Category 3 (Fast)	Speed $s_3 = 13.357$ Gflops Cost $c_{h,3} = \$0.370$ per hour
Datacenter	
Cost per month	$c_{h,DC} = \$0.022$ per GB
Data transfer cost	$c_{tsf} = \$0.055$ per GB
Bandwidth	
bw	125MBps

Table 2: Parameters of the IAAS Cloud platform.

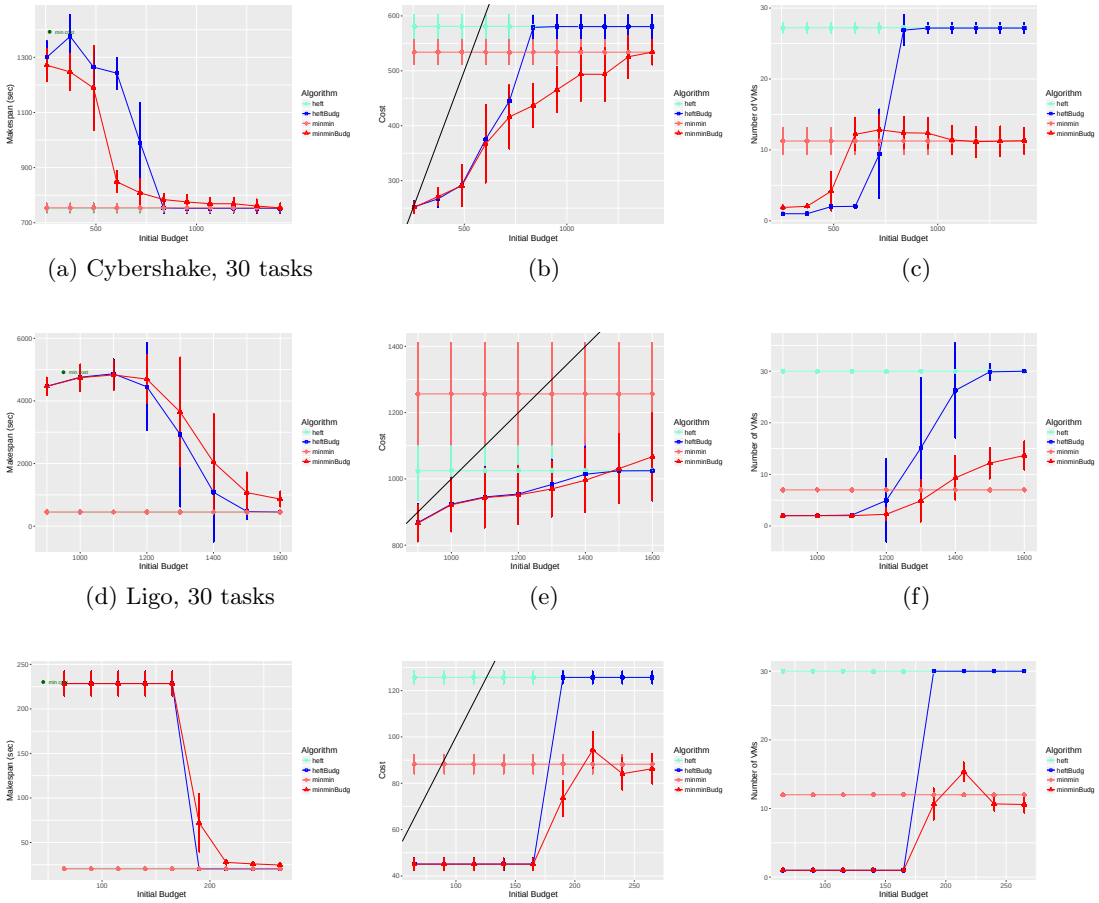
a Bag of Tasks, and the priority mechanism of HEFTBUDG becomes less useful. On the contrary, larger MONTAGE workflows keep numerous imbricated dependencies between tasks, and HEFTBUDG remains more efficient in terms of budget.

Regarding the behavior of our algorithms, increasing initial budget will lead to enrolling more VMs, with an exception: in Figures 2i, 3i and 4i, we see that the number of VMs can rise for intermediate values of budgets until exceeding the one of the baseline version, then decrease again until reaching it. This corresponds to the moment when several tasks have enough budget to leave their mid-efficient VM to go to a VM of the fastest category.

Next, we assess in Figures 6, 7 and 8 the impact of the amount of uncertainty in task weights. Each row represents results for one value of standard deviation σ used to generate task weights. We see that both HEFTBUDG and MIN-MINBUDG require a larger initial budget to achieve a given makespan, when σ increases; yet the budget constraint is respected, even in scenarios where task weights can be twice their mean value.

Finally we discuss the execution time of each algorithm. The experiments have been made on a computer with a Intel® Core™ i5-6200U CPU @ 2.30GHz × 4 processors. We recorded the time needed for each algorithm while calculating 5 continuous schedules, and executed 30 instances for each combination of parameters. We used three types of workflows (LIGO, MONTAGE, CYBERSHAKE) instantiated with 30, 60 and 90 tasks. As for the impact of the budget on the time needed to calculate a schedule, we used three characteristic values for each workflow, which we designate as "low", "high" and "medium". A "low" budget $\mathcal{B}_{min}$ corresponds to the minimum budget needed to find a schedule, a "high" one to a budget large enough to enroll an unlimited number of VMs. The "medium" budget is chosen as follows: for each workflow, we empirically find the minimum budget $\mathcal{B}_{\min_{best}}$ needed to obtain a makespan as good as the one found for baseline version of the algorithm, and take the average: $\mathcal{B}_{med} = \frac{\mathcal{B}_{\min_{best}} + \mathcal{B}_{min}}{2}$

Tables 3 and 4 show CPU times needed to calculate a schedule for workflows of varying type and size. For example, for a MONTAGE workflow of 90 tasks, HEFTBUDG needs 2.87 ± 0.52



(a) Cybershake, 30 tasks (b) (c)

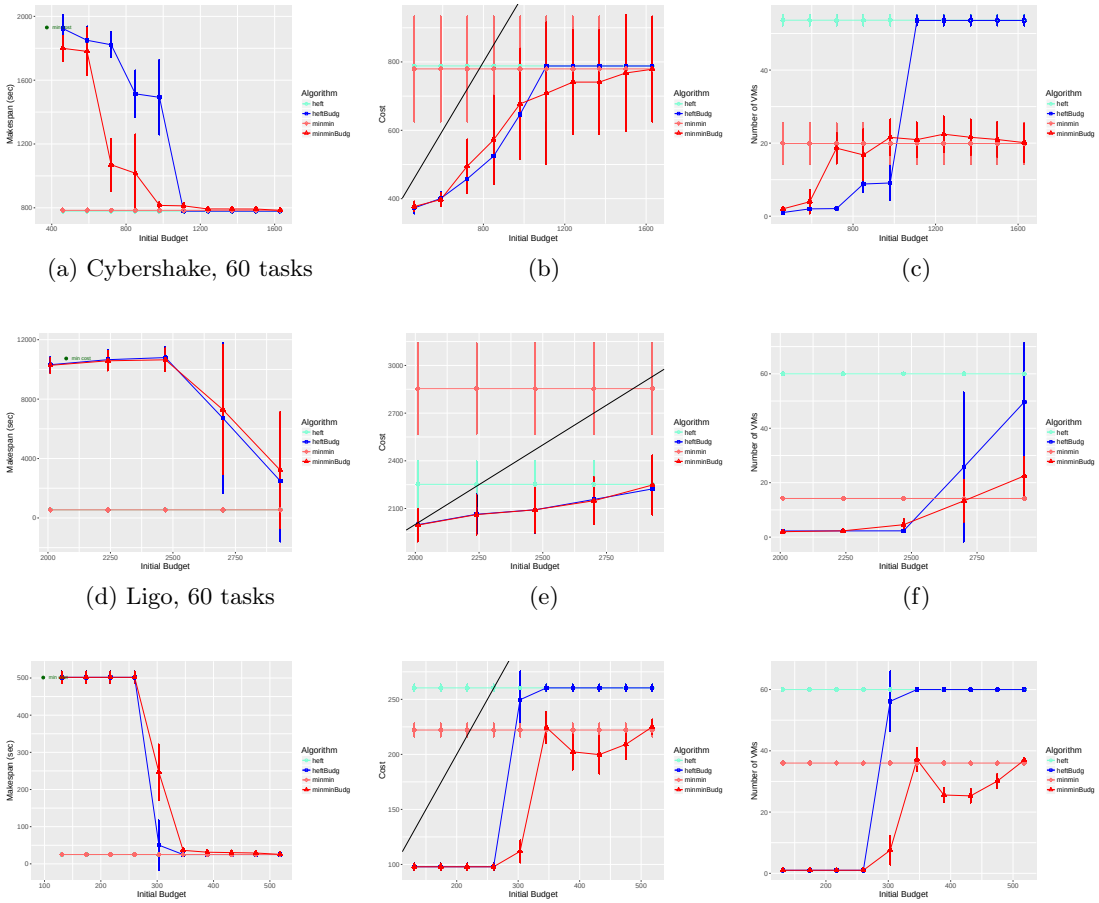
(d) Ligo, 30 tasks (e) (f)

(g) Montage, 30 tasks (h) (i)

Figure 2: HEFT, Min-Min, and their budget-aware extensions HEFTBudg and Min-MinBudg for the three workflow types with 30 tasks.

seconds to find a schedule when it only needs 0.60 ± 0.39 seconds for a Cybershake workflow or 0.72 ± 0.40 seconds for a Ligo workflow; such differences can be seen for the other algorithms as well.

Here are some concluding remarks about HEFTBudg and Min-MinBudg. To cope with uncertainty in task weights, we made a pessimistic estimation of the cost to transfer files, assuming they were always produced by another VM. This was safe but led to overestimating both the time and the budget for these transfers. Also, the budget assignment is somewhat unfair to the first scheduled tasks, which have no access to any leftover resource (the pot). In the next section, we aim at improving the schedule by re-examining the assignment and budget allotted to each task. We do this for HEFTBudg only, because it typically achieves a smaller makespan than Min-MinBudg for a prescribed budget. Of course, similar improvements could be designed for Min-MinBudg.



(a) Cybershake, 60 tasks (b) (c)

(d) Ligo, 60 tasks (e) (f)

(g) Montage, 60 tasks (h) (i)

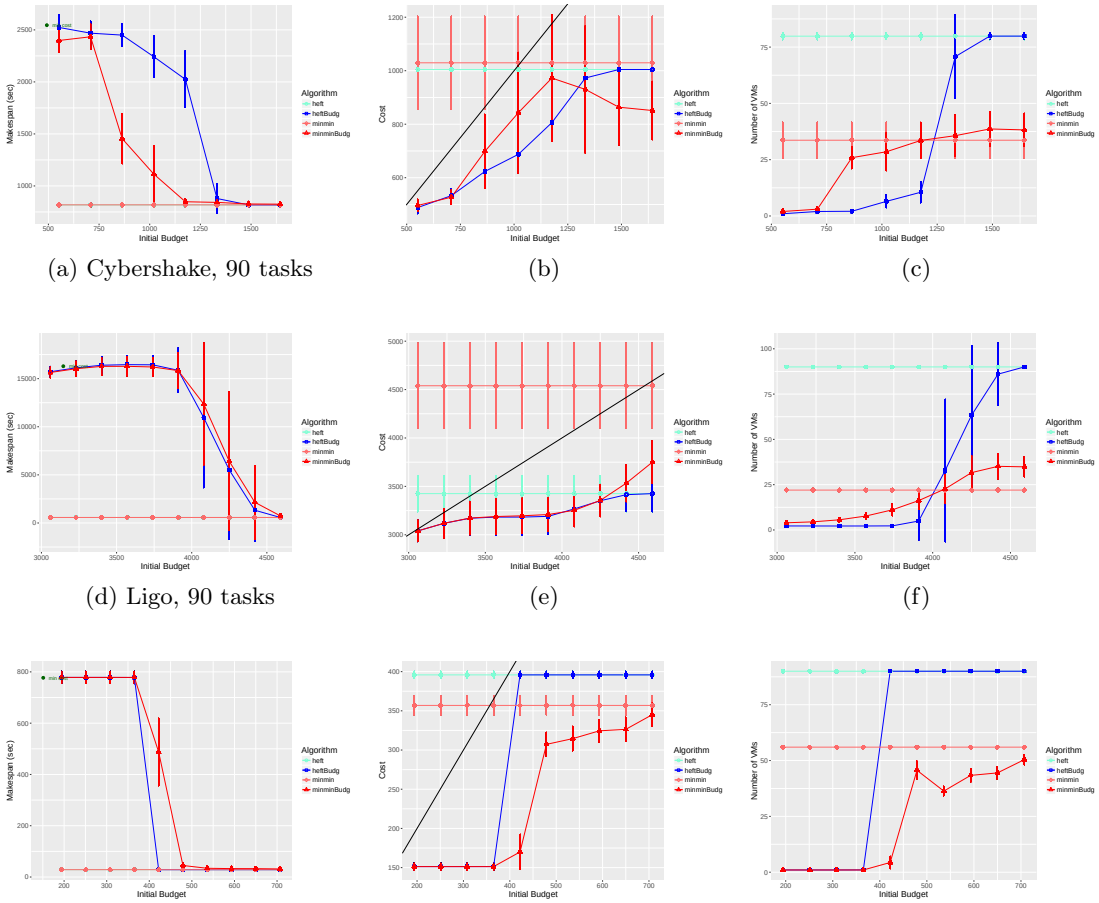
Figure 3: HEFT, Min-Min, and their budget-aware extensions HEFTBudg and Min-MinBudg for the three workflow types with 60 tasks.

5.3 Performance of HEFTBudg+ and HEFTBudg+Inv

We report in Figures 9, 10 and 11 the results obtained on objective values makespan, final cost and number of VMs as function of the initial budget for the algorithms HEFT, HEFTBudg, HEFTBudg+ and HEFTBudg+Inv used on Cybershake, Ligo and Montage workflows.

The schedules obtained for both refined algorithms HEFTBudg+ and HEFTBudg+Inv have a shorter makespan than HEFTBudg (see Figures 9, 10 and 11 a, d, g). Their makespan can be up to one third shorter than for HEFTBudg, *e.g.,* for Montage. Surprisingly the refined algorithms manage to achieve a smaller makespan using *fewer* VMs than HEFTBudg. This is mostly because they succeed in assigning interdependent tasks onto the same VM. The initial budget needed to obtain the same makespan as HEFT is the same for HEFTBudg, HEFTBudg+ and HEFTBudg+Inv. Moreover, the budget is respected overall, as was already the case with HEFTBudg.

To compare HEFTBudg+ and HEFTBudg+Inv, we observe that their makespans are very similar, apart from the case where a budget near the minimal one is needed to com-



(a) Cybershake, 90 tasks (b) (c)

(d) Ligo, 90 tasks (e) (f)

(g) Montage, 90 tasks (h) (i)

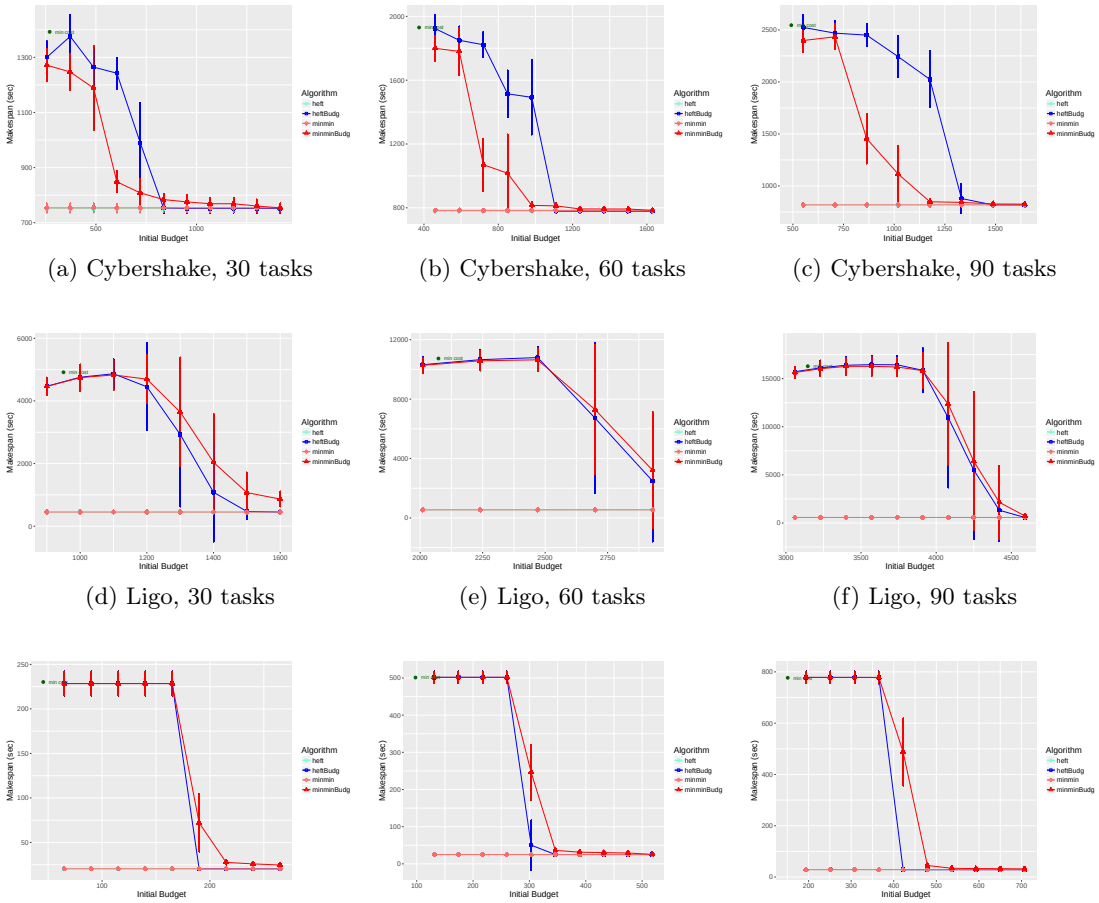
Figure 4: HEFT, Min-Min, and their budget-aware extensions HEFTBudg and Min-MinBudg for the three workflow types with 90 tasks.

plete a schedule: in that case, HEFTBudg+ obtains an average makespan twice shorter than HEFTBudg+Inv. This difference seen in this exact configuration may be due to the particular structure of Montage workflows: they have a lot of initial tasks, and the amount of work for every kind of tasks is of the same magnitude.

However, HEFTBudg+ and HEFTBudg+Inv achieve far better makespans than HEFTBudg but for a higher computational cost. For instance, for Montage with 90 tasks ànd a high budget, HEFTBudg finds a solution in 2.60 ± 0.28 seconds while HEFTBudg+ needs 379.45 ± 44.20 seconds, and HEFTBudg+Inv needs 382.29 ± 43.25 seconds.

6 Conclusion

We have presented a model and several budget-aware algorithms to schedule scientific workflows with stochastic task weights onto IaaS Cloud platforms. The first two algorithms, HEFTBudg and Min-MinBudg, are extensions of the well-known HEFT and Min-Min heuristics. We show that they manage to find a solution whose makespan remains simi-



(a) Cybershake, 30 tasks (b) Cybershake, 60 tasks (c) Cybershake, 90 tasks

(d) Ligo, 30 tasks (e) Ligo, 60 tasks (f) Ligo, 90 tasks

(g) Montage, 30 tasks (h) Montage, 60 tasks (i) Montage, 90 tasks

Figure 5: HEFT, Min-Min, and their budget-aware extensions HEFTBudg and Min-MinBudg for Cybershake, Ligo and Montage with various sizes.

lar to that of the baseline version while enforcing the prescribed budget. We observe that HEFTBudg obtains a better makespan than Min-MinBudg for a given budget, in particular for workflows with a non-trivial inter-dependency graph. We then propose two refined versions, HEFTBudg+ and HEFTBudg+Inv, that achieve better makespans, but at the cost of a higher CPU time.

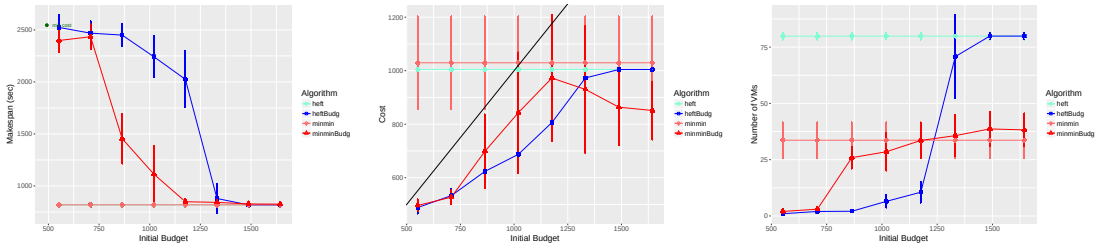
Further work will be devoted to extending the approach to on-line schedules, whenever the target Cloud infrastructure would allow to interrupt and re-schedule tasks on the fly. Indeed, if we monitor the execution of the tasks, we can detect unlikely events such as very long durations, and in such cases, it could be beneficial to interrupt some tasks and to re-schedule them onto faster VMs. Such dynamic decisions encompass risks in terms of both final makespan and budget. For instance, deriving execution timeouts is a challenging problem, but we hope to derive on-line heuristics that, with high probability, will decrease the final makespan while respecting the initial budget constraint.

	HEFT	Min-Min	HEFTBudg	Min-MinBudg
Low	2.66 ± 0.30 2.56	2.67 ± 0.31 2.58	2.60 ± 0.28 2.49	1.95 ± 0.17 1.90
Medium	2.67 ± 0.30 2.58	2.67 ± 0.30 2.58	2.59 ± 0.27 2.50	1.95 ± 0.16 1.90
High	2.68 ± 0.30 2.59	2.68 ± 0.31 2.57	3.42 ± 0.44 3.29	1.96 ± 0.17 1.92

Table 3: Time needed to calculate a schedule for each algorithm, for different budgets. Results are for a Montage workflow of 90 tasks and are given in seconds, in the form mean ± standard value, median.

	HEFT	Min-Min	HEFTBudg	Min-MinBudg
30	0.15 ± 0.03 0.15	0.11 ± 0.002 0.11	0.21 ± 0.003 0.21	0.089 ± 0.002 0.088
60	0.97 ± 0.01 0.97	0.90 ± 0.01 0.90	1.29 ± 0.02 1.29	0.65 ± 0.01 0.66
90	2.68 ± 0.30 2.59	2.68 ± 0.31 2.57	3.42 ± 0.44 3.29	1.96 ± 0.17 1.92
400	294.96 ± 15.83 297.23	395.80 ± 15.83 395.06	341.00 ± 14.64 340.15	268.15 ± 12.41 269.54

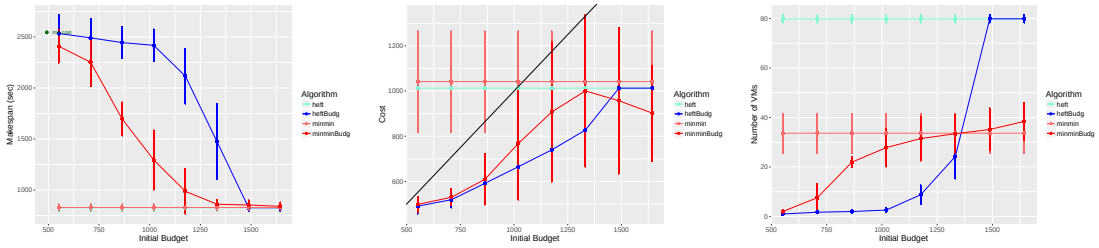
Table 4: Time needed to calculate a schedule for each algorithm, for a Montage workflow with 30, 60 and 90 tasks, and a high budget. Results are given in seconds, in the form mean ± standard value, median.



(a) Cyber, 90 tasks, $\sigma = 0.25$

(b)

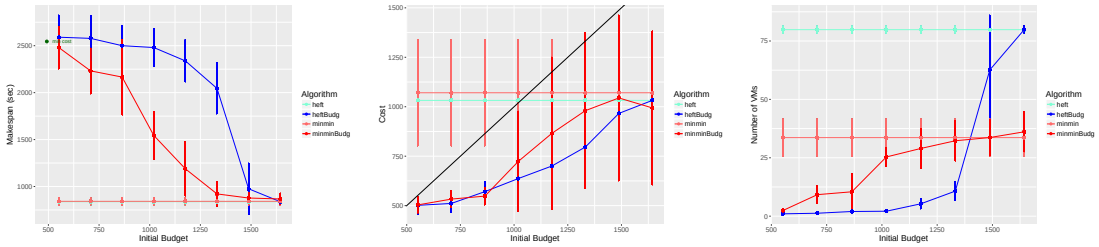
(c)



(d) Cyber, 90 tasks, $\sigma = 0.50$

(e)

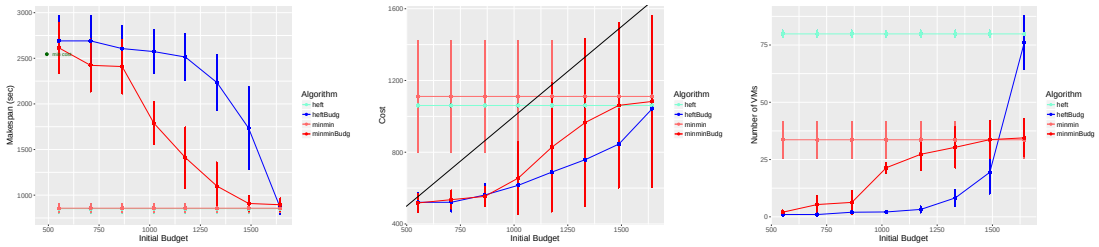
(f)



(g) Cyber, 90 tasks, $\sigma = 0.75$

(h)

(i)

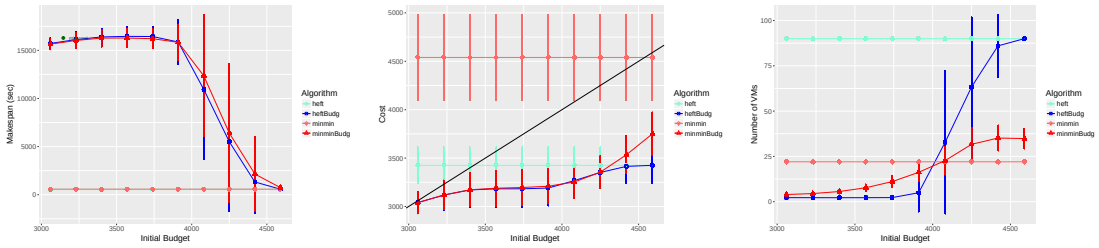


(j) Cyber, 90 tasks, $\sigma = 1.0$

(k)

(l)

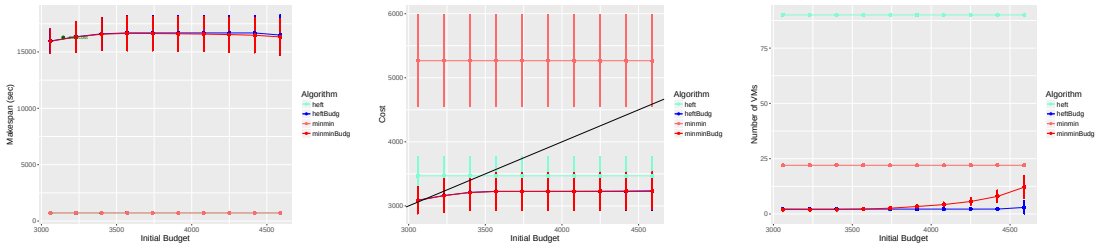
Figure 6: HEFT, Minmin, and their budgeted variants for different values of the standard deviation σ of task weights.



(a) Ligo, 90 tasks, $\sigma = 0.25$

(b)

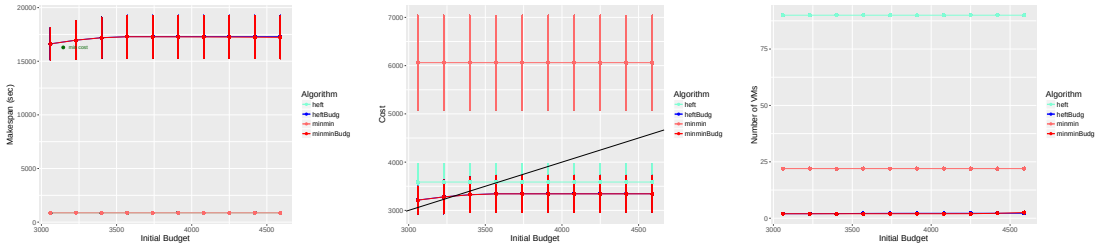
(c)



(d) Ligo, 90 tasks, $\sigma = 0.50$

(e)

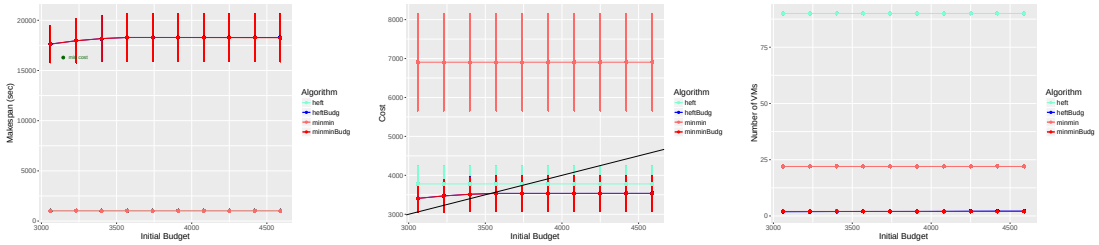
(f)



(g) Ligo, 90 tasks, $\sigma = 0.75$

(h)

(i)

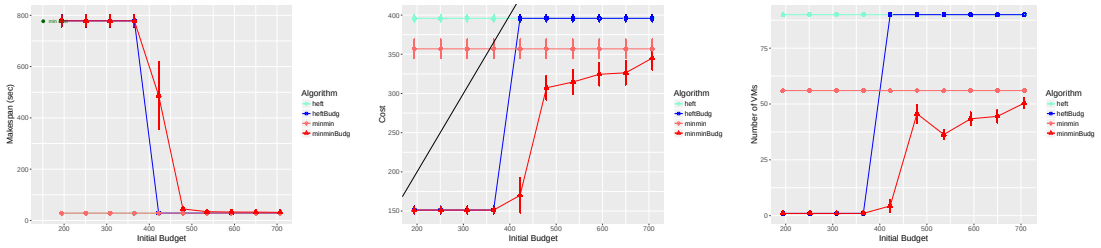


(j) Ligo, 90 tasks, $\sigma = 1.0$

(k)

(l)

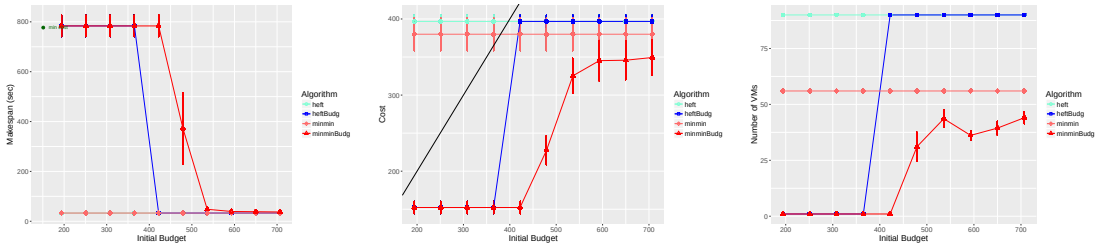
Figure 7: HEFT, Minmin, and their budgeted variants for different values of the standard deviation σ of task weights for LIGO type workflows.



(a) Montage, 90 tasks, $\sigma = 0.25$

(b)

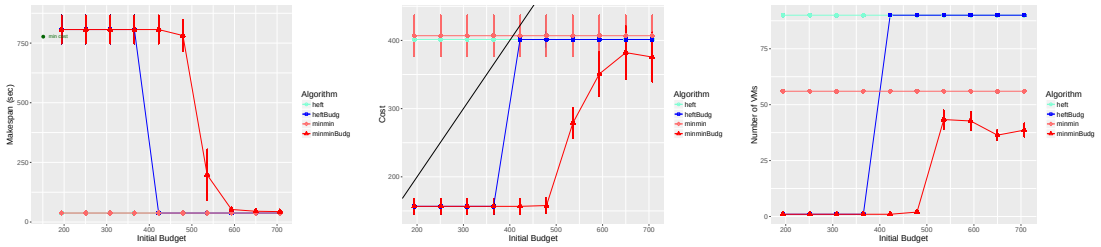
(c)



(d) Montage, 90 tasks, $\sigma = 0.50$

(e)

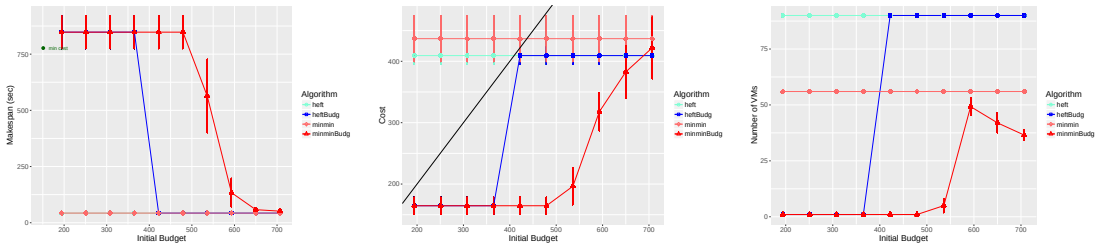
(f)



(g) Montage, 90 tasks, $\sigma = 0.75$

(h)

(i)

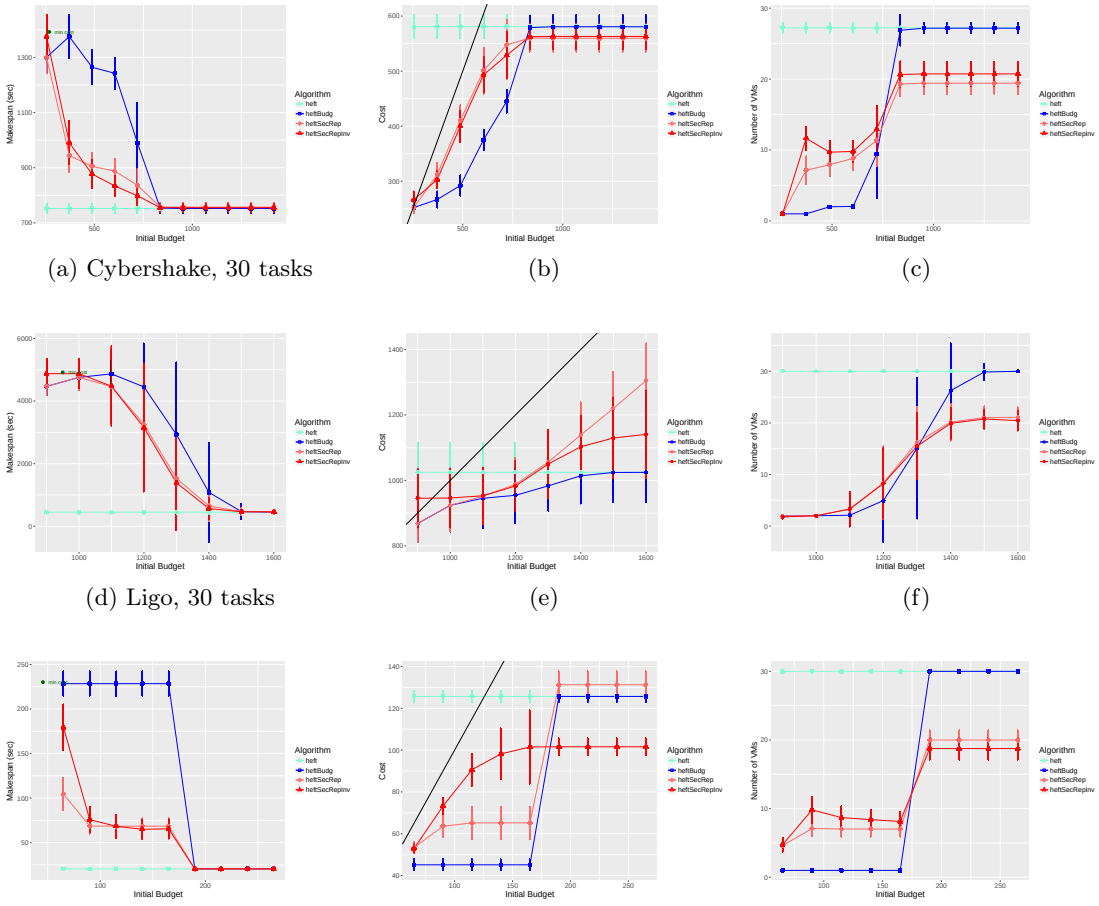


(j) Montage, 90 tasks, $\sigma = 1.0$

(k)

(l)

Figure 8: HEFT, Minmin, and their budgeted variants for different values of the standard deviation σ of task weights.

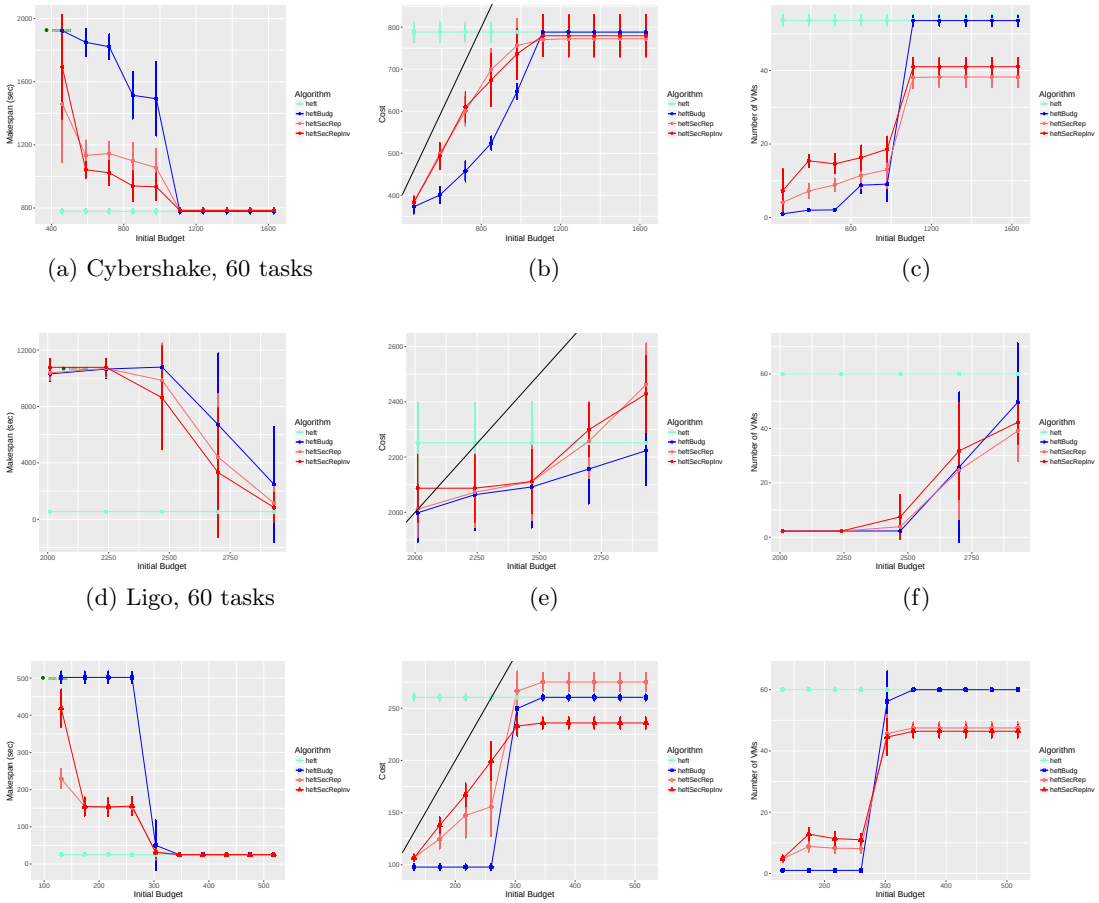


(a) Cybershake, 30 tasks (b) (c)

(d) Ligo, 30 tasks (e) (f)

(g) Montage, 30 tasks (h) (i)

Figure 9: HEFTBUDG+, HEFTBUDG+INV compared to HEFT and HEFTBUDG for the three workflow types with 30 tasks.

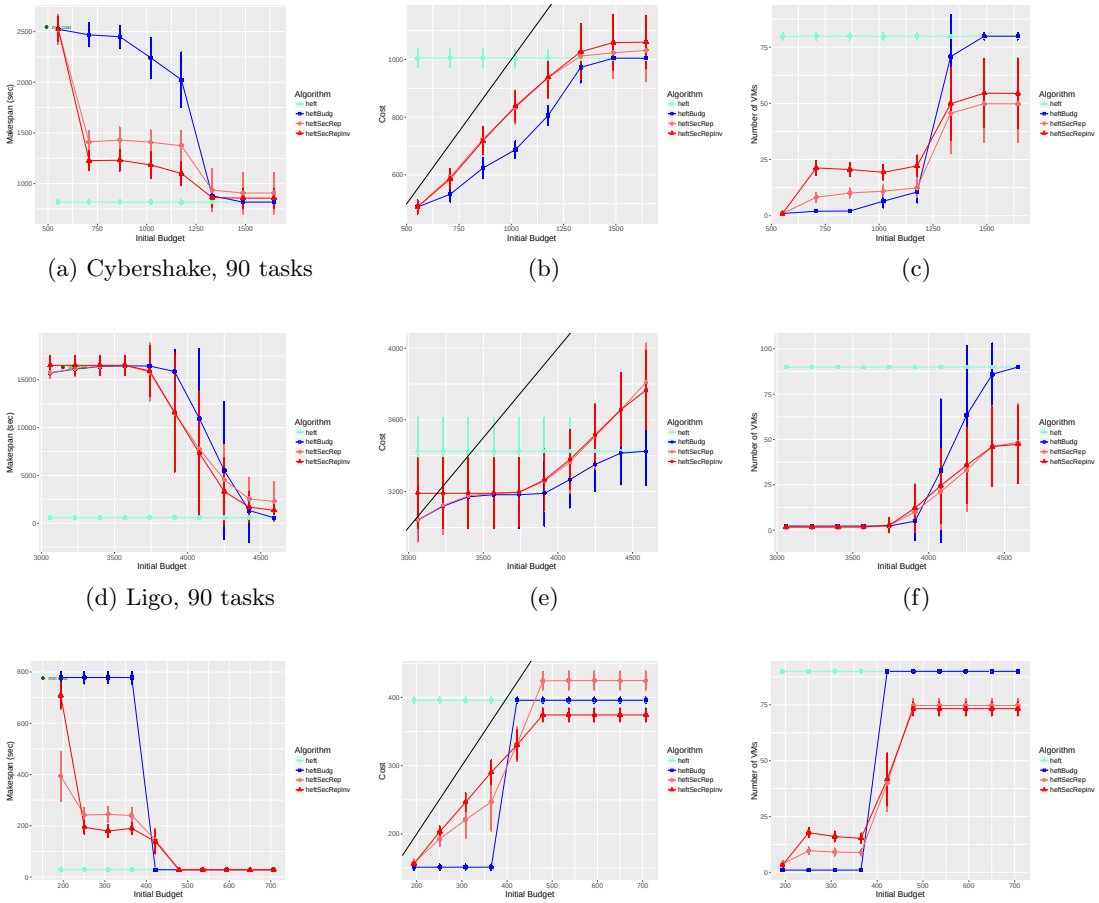


(a) Cybershake, 60 tasks (b) (c)

(d) Ligo, 60 tasks (e) (f)

(g) Montage, 60 tasks (h) (i)

Figure 10: HEFTBUDG+, HEFTBUDG+INV compared to HEFT and HEFTBUDG for the three workflow types with 60 tasks.



(a) Cybershake, 90 tasks (b) (c)

(d) Ligo, 90 tasks (e) (f)

(g) Montage, 90 tasks (h) (i)

Figure 11: HEFTBUDG+, HEFTBUDG+INV compared to HEFT and HEFTBUDG for the three workflow types with 90 tasks.

References

[1] E. N. Alkhanak, S. P. Lee, R. Rezaei, and R. M. Parizi. Cost optimization approaches for scientific workflow scheduling in cloud and grid computing: A review, classifications, and open issues. *Journal of Systems and Software*, 113:1–26, 2016.

[2] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. Characterization of scientific workflows. In *SC'08 Workshop: The 3rd Workshop on Workflows in Support of Large-scale Science (WORKS08) web site*, Austin, TX, Nov. 2008. ACM/IEEE.

[3] T. Braun, H. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A. Reuther, J. P. Robertson, M. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001.

[4] E. Caron, F. Desprez, T. Glatard, M. Ketan, J. Montagnat, and D. Reimert. Workflow-based comparison of two distributed computing infrastructures. In *Workflows in Support of Large-Scale Science (WORKS10)*, New Orleans, November 14 2010. In Conjunction with Supercomputing 10 (SC'10), IEEE. hal-00677820.

[5] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *J. Parallel Distributed Computing*, 74(10):2899–2917, 2014.

[6] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. Workflow Management in Condor. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 357–375. Springer, 2007.

[7] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[8] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46(0):17–35, 2015.

[9] P. Ezzatti, M. Pedemonte, and A. Martín. An efficient implementation of the min-min heuristic. *Comput. Oper. Res.*, 40(11), 2013.

[10] G. Juve, A. L. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi. Characterizing and profiling scientific workflows. *Future Generation Comp. Syst.*, 29(3):682–692, 2013.

[11] C. Lin and S. Lu. Scheduling scientific workflows elastically for cloud computing. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 746–747. IEEE, 2011.

[12] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski. Algorithms for cost- and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. *Future Generation Computer Systems*, 48:1–18, 2015.

[13] M. Mao and M. Humphrey. Scaling and scheduling to maximize application performance within budget constraints in cloud workflows. In *Parallel and Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 67–78. IEEE, 2013.

[14] Pegasus. Pegasus workflow generator. `https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator`, 2014.

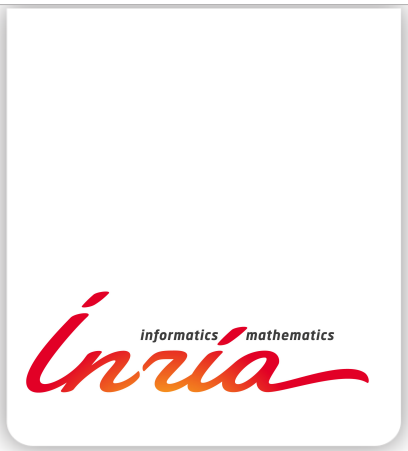
[15] SimDag. Programming environment for DAG applications. `http://simgrid.gforge.inria.fr/simgrid/3.13/doc/group__SD__API.html`, 2017.

[16] S. Smanchat and K. Viriyapant. Taxonomies of workflow scheduling problem and techniques in the cloud. *Future Generation Computer Systems*, 52:1–12, 2015.

[17] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Systems*, 13(3):260–274, 2002.

[18] D. Yuan, Y. Yang, X. Liu, and J. Chen. A data placement strategy in scientific cloud workflows. *Future Generation Comp. Syst*, 26(8):1200–1214, 2010.

[19] L. Zeng, B. Veeravalli, and X. Li. SABA: A security-aware and budget-aware workflow scheduling strategy in clouds. *J. Parallel Distrib. Comput*, 75:141–151, 2015.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399